1. Tempo disponibile 120 minuti (90 minuti per gli studenti di "Introduzione agli Algoritmi" - 6 CFU, che devono fare solo i primi 3 esercizi).
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo MYSTERY:

Algorithm 1: MYSTERY(INT n) $\rightarrow$ INT

```
i ← 1
j ← 0
while i ≤ n do
|  i ← i × 2
|  j ← j + MYSTERY2(n/2) + MYSTERY2(n/2)
end
return j

function MYSTERY2(INT m) → INT
if m ≤ 1 then
|  return 1
else
|  j ← 1
|  while j ≤ m do
|  |  j ← j + 1
|  end
|  return j + MYSTERY2(m/3) + MYSTERY2(m/3)
end
```

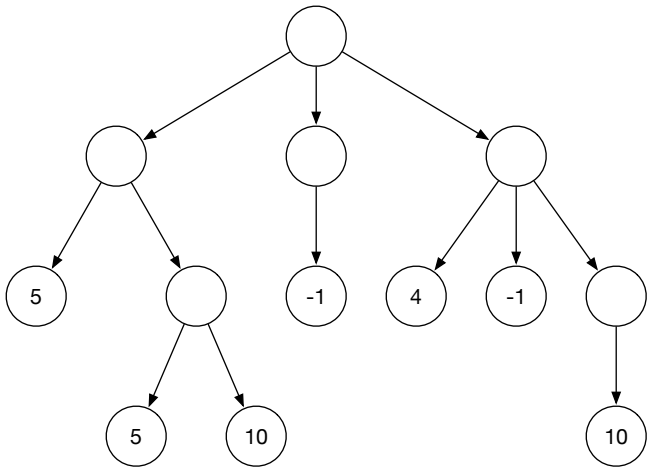
Soluzione. Iniziamo dall'analisi del costo computazionale $T'(m)$ della funzione MYSTERY2. Si tratta di una funzione che soddisfa la seguente relazione di ricorrenza:

$$T'(m) = \begin{cases} 1 & \text{se } m = 1 \\ 2 \times T'(\frac{m}{3}) + m & \text{altrimenti} \end{cases}$$

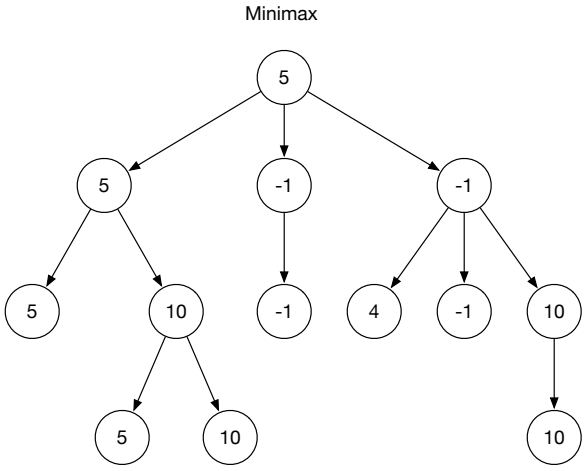
in quanto la funzione è ricorsiva con costo costante nel caso base e, nel caso ricorsivo, con due chiamate ricorsive in cui si riduce di 3 volte il parametro più un ciclo di costo lineare. Tale relazione di ricorrenza può essere risolta utilizzato il Master Theorem considerando i parametri $a = 2$, $b = 3$ e $\beta = 1$ che implicano $\alpha = \frac{\log 2}{\log 3} = \frac{1}{\log 3}$. Avendo $\alpha < \beta$ ci troviamo nel caso 3 del teorema che implica $T'(m) = \Theta(m)$.

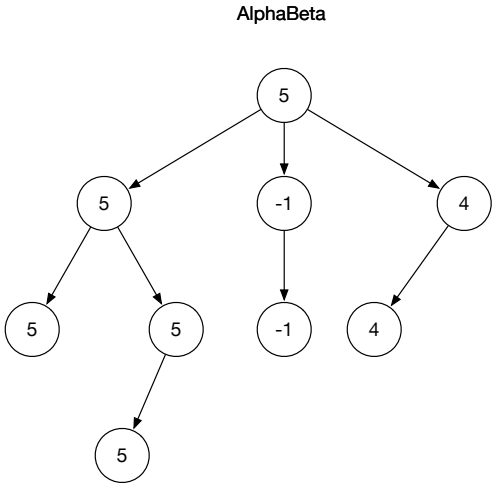
Passiamo ora all'analisi del costo computazionale $T(n)$ della funzione MYSTERY che esegue per $\log n$ volte un ciclo (in quanto ad ogni loop si raddoppia l'indice di controllo del ciclo) che contiene due chiamate alla funzione MYSTERY2 con parametro $n/2$. Abbiamo quindi $T(n) = \log n \times 2 \times T'(n/2) = \log n \times 2 \times \Theta(n/2) = \Theta(\log n \times n)$, che solitamente scriviamo come $\Theta(n \log n)$.

2. Dato il seguente albero, con label numeriche assegnate alle foglie, mostrare le label dei nodi intermedi dopo l'applicazione degli algoritmi Minimax e AlphaBeta pruning assumendo che la radice dell'albero sia un nodo di **massimizzazione** (i.e. prende il valore massimo tra i valori assegnati ai figli diretti). Nel disegnare l'albero dopo l'applicazione dell'AlphaBeta pruning non mostrare i rami dell'albero non visitati.



Soluzione. Gli alberi risultanti dopo l'applicazione dell'algoritmo Minimax e AlphaBeta pruning sono mostrati di seguito.





3. Progettare un algoritmo che dato un vettore di numeri $V[1..n]$ stampa una sequenza di indici $j_1, j_2, \ldots, j_k$, con k più grande possibile, tale che $j_i < j_{i+1}$ e $V[j_i] \geq V[j_{i+1}]$, per ogni $i \in \{1 \ldots k-1\}$.

Soluzione. È possibile utilizzare la programmazione dinamica considerando i problemi $P(i)$, con $i \in \{1 \ldots n\}$, tale che $P(i)$ = sequenza massima di indici $j_1, j_2, \ldots, j_k \in \{1 \ldots i\}$, tale che $j_l < j_{l+1}$ e $V[j_l] \geq V[j_{l+1}]$, per ogni $i \in \{1 \ldots k-1\}$, ed inoltre $j_k = i$. Si noti quest'ultima condizione: nel problema $P(i)$ consideriamo sequenze che terminano in posizione i.

Tali problemi $P(i)$ possono essere risolti considerando che:

$$P(i) = \begin{cases} 1 & \text{se } i = 1 \\ 1 + max\{P(j) \mid 1 \leq j < i \text{ and } V[j] \geq V[i]\} & \text{se } i > 1 \end{cases}$$

Per stampare la sequenza di indici più lunga si utilizza un array ausiliario *prec*, tale che *prec*$[i]$ conterrà, per ogni $i \in \{1 \ldots n\}$, il penultimo indice della sottosequenza ottimale per il problema $P(i)$. Gli indici della sequenza massima possono quindi essere reperiti a ritroso partendo dall'indice finale della sequenza più lunga.

Cont.

Algorithm 2: MassimaSequenza(Int $V[1..n]$)

```
// Inizializzazione
Int prec[1..n], A[1..n], maxIndice, precIndice
A[1] ← 1; prec[1] ← −1; maxIndice ← 1      // maxIndice indicherà il j_k della sequenza massima
// Risoluzione sottoproblemi P(i)
for i ← 2 to n do
    // Ricerca del penultimo indice della sequenza del problema P(i)
    precIndice ← −1
    for j ← 1 to i − 1 do
        if V[j] ≥ V[i] and (precIndice = −1 or A[j] > A[precIndice]) then
            precIndice ← j
        end
    end
    // aggiornamento strutture dati
    if precIndice = −1 then
        A[i] ← 1
    else
        A[i] ← A[precIndice] + 1
    end
    prec[i] ← precIndice
    if A[i] > A[maxIndice] then
        maxIndice ← i
    end
end
// stampa della sequenza di lunghezza massima
printPath(prec, maxIndice)

// funzione ausiliaria di stampa del cammino
procedure printPath(Int prec[1..|V|], Int i)
if i ≠ −1 then
    printPath(prec, prec[d])
    print i
end
```

L'Algoritmo 2 risolve i problemi $P(i)$ inserendo le relative soluzioni nell'array $A[1..n]$, e poi procede per la stampa della sequenza di lunghezza massima trovata. Il costo computazionale $T(n)$ di tale algoritmo è dato dai due cicli **for** annidati in quanto le altre operazioni impiegano tempo costante e gli altri cicli sono eseguiti una quantità inferiore di volte. Il ciclo interno dei due **for** annidati viene eseguito un numero di volte pari a $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$ che implica $T(n) = \Theta(n^2)$.

Cont.

4. Una comunità di formiche deve costruire più velocemente possibile un formicaio che permetta alle formiche di spostarsi fra un certo insieme di stanze S. Ogni stanza $s \in S$ richiede un tempo di costruzione $K(s)$. Per ogni coppia di stanze $s_1, s_2 \in S$, esiste un tempo $T(s_1, s_2)$ per costruire una galleria di collegamento diretto tra le due stanze (si ha che $T(s_1, s_2) = T(s_2, s_1)$). Il tempo complessivo di costruzione del formicaio è dato dalla somma dei tempi di costruzione delle gallerie più la somma dei tempi di costruzione delle stanze. Il formicaio non deve obbligatoriamente avere una galleria diretta fra ogni coppia di stanze, ma è sufficiente che per ogni coppia di stanze esista un cammino di collegamento, che potrebbe includere più gallerie e stanze intermedie. Progettare un algoritmo che dato l'insieme di stanze S, la funzione $K : S \to \mathbb{R}$ (tale che $\forall s \in S \,.\, k(s) \geq 0$), e la funzione $T : S \times S \to \mathbb{R}$ (tale che $\forall s_1, s_2 \in S \,.\, T(s_1, s_2) \geq 0$), restituisce il tempo minimo di costruzione del formicaio.

   Soluzione. Il problema può essere interpretato come un problema su grafi, dove le stanze rappresentano i vertici, le gallerie gli archi, ed i tempi di costruzione delle gallerie i pesi degli archi. Abbiamo in input un grafo completo (quindi con un numero di archi quadratico rispetto al numero di vertici), in quanto esiste una potenziale galleria per ogni coppia di stanze. Le gallerie che minimizzano i tempi di costruzione coincidono con un minimum spanning tree per tale grafo. Una volta trovato il tempo complessivo necessario per costruire le gallerie del minimum spanning tree, si aggiunge il tempo di costruzione delle stanze. L'Algoritmo 3 utilizza l'algoritmo di Prim prendendo in input l'insieme S, e due strutture K e T che consideriamo essere rispettivamente un array ed una tabella che utilizza gli elementi di S come indici (a titolo di esempio, usiamo $T[s, v]$, con $s, v \in S$, per indicare il tempo di costruzione della galleria dalla stanza s alla stanza v). Anche la struttura dati ausiliaria $D[S]$ è un array indicizzato su S. Questa è una pseudo-notazione di comodo: una vera implementazione potrebbe usare la convenzione che le stanze vengono rappresentate dall'insieme di indici $\{1 \ldots |S|\}$. Durante il calcolo del minimum spanning tree, ad ogni selezione di una galleria, si incrementa un contatore *tot* con il relativo tempo di costruzione. Al termine del calcolo del minimum spanning tree, si aggiungono a *tot* i tempi di costruzione delle stanze. Infine, si restituisce il contatore *tot*. Il costo computazionale corrisponde con il costo dell'algoritmo di Prim che, considerando il grafo completo, può essere espresso come $T(n) = O(n^2 \log n)$, dove $n = |S|$ è il numero delle stanze.

Algorithm 3: FORMICAIO(SET S, NUMBER[S] K, NUMBER[S, S] T) $\rightarrow$ NUMBER

```
/* inizializzazione strutture dati                                    */
INT v, u
NUMBER D[S], tot
SET visited
MINPRIORITYQUEUE[S, NUMBER] Q ← new MINPRIORITYQUEUE[S, NUMBER]()
s ← S.selectOneElement()
D[s] ← 0; Q.insert(s, D[s]); visited ← {s}
for v ∈ S \ {s} do
  | D[v] ← T[s, v]; Q.insert(v, D[v])
end

/* esecuzione algoritmo di Prim (modificato)                          */
tot ← 0
while not Q.isEmpty() do
  | u ← Q.findMin(); Q.deleteMin(); visited ← visited ∪ {u}
  | tot ← tot + D[u]
  | for v ∈ S \ visited do
  |   | if T[u, v] < D[v] then
  |   |   | /* scoperta di una galleria migliore per raggiungere v  */
  |   |   | Q.decreaseKey(v, D[v] − T[u, v])
  |   |   | D[v] = T[u, v]
  |   | end
  | end
end

/* aggiunta dei tempi di costruzione delle stanze                     */
for v ∈ S do
  | tot ← tot + K[v]
end

return tot
```

The End.